OpenCL-based Design Methodology for Application-Specific Processors

Pekka O. Jääskeläinen*, Carlos S. de La Lama†, Pablo Huerta† and Jarmo H. Takala*
*Tampere University of Technology, Department of Computer Systems, Tampere, Finland
Email: {pekka.jaaskelainen, jarmo.takala}@tut.fi
†Universidad Rey Juan Carlos, Department of Computer Architecture, Móstoles, Madrid, Spain
Email: {carlos.delalama, pablo.huerta}@urjc.es

***Abstract*—OpenCL is a programming language standard which enables the programmer to express the application by structuring its computation as *kernels*. The OpenCL compiler is given the explicit freedom to parallelize the execution of *kernel instances* at all the levels of parallelism. In comparison to the traditional C programming language which is sequential in nature, OpenCL enables higher utilization of parallelism naturally available in hardware constructs while still having a feasible learning curve for engineers familiar with the C language.**

This paper describes methodology and compiler techniques involved in applying OpenCL as an input language for a design flow of application-specific processors. At the core of the methodology is a whole program optimizing compiler that links together the host and kernel codes of the input OpenCL program and parallelizes the result on a customized statically scheduled processor. The OpenCL vendor extension mechanism is used to provide clean access to custom operations.

The methodology is studied with a design case to verify the scalability of the implementation at the instruction level and to exemplify the use of custom operations. The case shows that the use of OpenCL allows producing scalable application-specific processor designs and makes it possible to gradually reach the performance of hand-tailored RTL designs by exploiting the OpenCL extension mechanism to access custom hardware operations of varying complexity.

***Index Terms*—OpenCL, Application-Specific Processors, Hardware accelerators, Instruction level parallelism, VLIW, Transport Triggered Architectures**

I. Introduction

Today the trend in computation platform design is to add more independent processor cores and processing elements to improve throughput by means of parallel execution instead of increasing the clock frequency and the level of pipelining of single monolithic cores. This is due to the fact that the clock frequencies of processor designs are reaching the limits of the CMOS technology and microarchitecture designs [1]. In addition, high clock frequencies might lead to heat problems due to higher power consumption [2].

Programming such parallel processors requires programming languages supporting parallelism. The C programming language was developed in 1970s and has retained its popularity ever since [3]. It has been the traditional choice especially for embedded systems engineers due to its "close-to-hardware" nature and widely available compiler support. In addition to programming existing processors, the C language has been widely used as an input to hardware design flows. Especially so called hardware/software co-design toolsets often start from application descriptions in C which are then gradually converted, automatically or manually, to hardware accelerators or customized processors executing the described algorithm faster than an off-the-shelf processor would. However, as C is a sequential programming language with unrestricted pointers and no standardized means to describe parallel execution at the multiple levels of parallelism, its capabilities in the generation of hardware accelerators with adequate throughput are limited.

Experience has shown that it is very difficult, computationally expensive, and often just plain impossible to extract parallelism from sequentially defined programs [4].

OpenCL standard [5] sidesteps this issue by structuring computation into kernels, and specifying that there are no dependences between kernel instances by default. The implementation is free to execute code from the different "kernel instances" sequentially, in parallel, or in an interleaved fashion, as long as the synchronization primitives (barriers) present in the kernel descriptions are respected. This freedom is utilized in our methodology by extracting instruction level parallelism from the kernel instances to improve the utilization of the available hardware resources in the automatically generated statically scheduled processor architecture. In addition to the parallel execution, an important feature of the proposed design flow is a clean and simple way to use custom hardware operations from the OpenCL C kernels using the OpenCL extension API.

This paper proposes methodology using OpenCL as an input language for designing application-specific processor (ASP) based hardware accelerators. At the core of this work is a compilation algorithm that allows full program offline compilation of OpenCL applications, including both the host program and the kernels, together to a single processor binary that is executable on a standalone customized processor. The use of OpenCL alleviates the exploration of design tradeoffs between silicon area and execution performance.

The paper is organized as follows. Related work is reviewed in Section II. Section III introduces the OpenCL standard and its benefits when used in ASP programming. Section IV

This research was partially funded by the Academy of Finland. The 1st author was supported financially by the Foundation of Nokia Corporation.

describes the processor architecture template and the toolset used as the framework for the automated generation of ASPs. Section V presents the practical issues in compiling OpenCL programs fully offline to produce scalable parallel implementations of the accelerated algorithms. Section VI presents proof-of-concept experiments of the methodology, and finally the paper is concluded in Section VII.

II. Related Work

In general, support for OpenCL has been increasingly started to appear from major companies such as Apple, NVIDIA, AMD, Intel, and S3. Thus, it seems OpenCL is here to stay and an important standard to support in the future.

Recently there has appeared a few publications on using GPGPU programming paradigms for generating code for non-GPU devices. The papers that have been published describe the use of the proprietary CUDA [6] language as the input while our work is based on the standardized OpenCL. However, as OpenCL and CUDA are very similar we consider these projects related to ours.

MCUDA [7] is a framework that aims to replace the sub-optimal CUDA to x86 compilation tool of the NVIDIA SDK with a version that parallelizes the execution on multiple host cores. The framework creates loops out of multiple work-item execution to retain work group barrier semantics. However, the parallelization is considered only at the task level while in our work focus is on instruction level parallelization issues.

FCUDA [8] is a source-to-source translator built on techniques implemented in MCUDA. They use the AutoPilot tool from AutoESL [9] for high level synthesis. Their main focus is on task level parallelization while leaving the important instruction level parallelism between work-items to lesser attention. Exploiting the ILP within a single wide statically scheduled core has its benefits as there are less off-core synchronization and communication required because more of the shared variables between work items can be stored in fast general-purpose registers.

CUDA is used as a starting point for hardware accelerator generation for FPGAs in [10]. The approach is exemplified with a kernel used to implement the *MrBayes* algorithm. The paper shows a procedure on how to map the relatively simple kernel of this algorithm to a pipelined hardware design. The approach differs from ours mainly in the reprogrammability. While our approach is based on a processor template with simplistic control logic, their approach generates directly hardware constructs with schedules implemented as state machines. This approach might lead to complex state machines when the mapped kernel is not trivial and contains control flow. In addition, the approach they present is not automatized while our contribution is to present a fully operational tool flow that targets a customizable processor architecture template, thus enables scaling of datapath resources according to the ILP available in the compiled OpenCL kernels.

III. Open Computing Language

OpenCL (Open Computing Language) [5] is a standard for programming heterogeneous multiprocessor platforms. The standard defines a C language API for invoking "kernels" (functions describing parallel execution on the device) and a C-based language called OpenCL C that is used for defining the kernels.

Albeit the background of OpenCL is clearly in the general-purpose computing on graphics processing units (GPGPU) community, and it is closely resembling the proprietary CUDA language from NVIDIA [6], the aim of OpenCL is to become a universal language for programming platforms with heterogeneous processing devices such as GPUs, CPUs, DSPs, etc.

What makes OpenCL an attractive candidate to act as an input for customized processor design flow is that it allows explicit definition of parallel execution at multiple levels. Operations on its vector data types invoke data level parallelism within a single kernel instance while the kernel instances itself implicitly describe parallel execution which is explicitly synchronized with barriers. The host API allows describing the number of "work-items" (instances of kernels executed in parallel) in a number of "work-groups", and the compiler and the execution platform is left the freedom - and the responsibility - to actually map the descriptions to the underlying hardware as efficiently as possible.

For our work, the interesting aspect of OpenCL is that it opens the possibility of design space exploration of the execution platform's area/performance ratio by means of allowing the scaling the performance by adding or removing computational resources according to the number of work items to be executed in parallel.

As a language for implementing processing kernels such as DSP filters as hardware accelerators, OpenCL C is clearly more powerful than the traditional C. While providing the most useful characteristics of C, the following differences and additional features stand out:

- *Implicit independence between work-items and work-groups.* As the execution is assumed to be independent, including memory accesses not only to "private" storage but also to shared "local" and "global" memory, it is possible to parallelize code from multiple work-items at the different granularities of parallelism. All synchronization is done by explicit barrier and memory fence calls.
- *Support for multiple disjoint address spaces* helps in alias analysis and enables explicit access to multiple separate memories.
- *No dynamic memory allocation.* The data memory consumption of the kernels can be estimated at compile time.
- *Vector data types.* Allows defining vector computation which can be trivially parallelized at instruction and data levels.
- *Recursion not supported.* Enables aggressive procedure call inlining.

IV. Transport Triggered Architectures

In this work, we have used transport triggered architecture (TTA) as the processor template. TTA reminds VLIW architectures [11] and the main difference between TTAs and VLIWs can be seen in how they are programmed: instead of defining

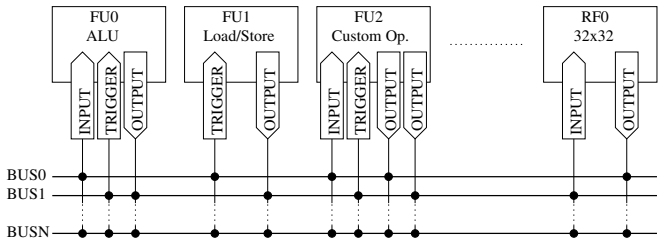

Fig. 1. Example of a TTA processor.

which operations are started in which function units (FU) at which instruction cycles, TTA programs are defined as data transports between register files (RF) and FUs of the datapath. The operations are started as side-effects of writing operand data to the "triggering port" of the FU. Fig. 1 presents a simple example TTA processor. The modularity of TTA enables easy customization of processor designs, making it an interesting architecture template for automated processor generation.

Thanks to its programmer-visible interconnection network, TTA datapath can support more FUs with simpler RFs [12] than an operation-programmed VLIW can. Because the scheduling of data transports between datapath units are programmer-defined, there is no obligation to scale the number of RF ports according to the number of FUs [13]. In addition, the datapath connectivity can be tailored according to the application at hand, adding only the bypassing paths that benefit the application the most potentially improving the maximum clock frequency.

TTA-based Codesign Environment (TCE) [14] is a processor design toolset that provides a complete design flow from software written in C or C++ down to parallel TTA program image and VHDL implementation of the processor. In this paper we explore the benefits of OpenCL as an additional input language alternative for TCE.

Because TTA is a statically scheduled architecture with low level details of execution exposed to the programmer, the runtime efficiency of the end results produced with the design toolset depends heavily on the quality of the compiler. The TCE complier uses the LLVM compiler infrastructure [15] for the frontend, the global middle-end optimizations (such as aggressive inlining and dead code elimination), and parts of the backend (the instruction selector and the register allocator). LLVM has been also used to implement the OpenCL transformation algorithms presented later in the paper. The final phases of TCE code generation have been written from the scratch to provide efficient retargetable instruction scheduling and TTA-specific optimizations.

V. Compiling OpenCL for Application-Specific Processors

The portability of OpenCL programs allows development and verification of the application code first outside the TCE toolset and later the code can be recompiled using the TCE compiler to generate code for a TTA-based ASP. Thus, the proposed OpenCL to ASP design methodology usually starts from implementation and verification of the OpenCL application using, for example, a GPU-based compilation environment and continues using the TCE tools for co-design of the ASP that can execute the application as efficiently as possible.

The core algorithms and concepts used in efficient compilation of OpenCL kernels to instruction level parallel code are described in Subsections V-A-V-D. Subsection V-E describes the way we use OpenCL API to let the hardware designer to access "custom operations" or "special function units" in the underlying ASP design.

A. Standalone Execution of OpenCL Applications

OpenCL is a computing language that is primarily meant for programming heterogenous multicore platforms. However, as one of the goals of OpenCL is to enable portability across multiple platforms, it is possible to execute full OpenCL programs purely using a single processor. This notion leads to two different setups for the generated ASPs:

1) *Standalone.* The ASP executes both the OpenCL host and device code. In this mode, the compiler compiles and links both the host and kernel programs together to a single processor binary that is executable on a standalone customized processor. No OpenCL support is required from the (possible) host processor of the ASP. However, the whole source code of the kernel must be available for offline compilation unless the ASP also includes an OpenCL C compiler, which is usually unrealistic.
2) *Host/device.* The ASP executes only the kernels implemented with OpenCL C and is commanded by a host processor. This is the standard CPU/GPU setup and requires OpenCL runtime and platform APIs to be implemented in the host. Supports also kernel code manipulation in runtime as the kernels can be recompiled on the host.

In our experiments, we used the standalone setup to produce standalone ASPs. Thus, in the terms of the OpenCL platform model, the generated ASPs act as the *host*, the *compute device*, and the *compute unit* at the same time. Function units of the TTA can be considered to be *processing elements*. In the terms of OpenCL memory model, the *global memory* and the *constant memory* can be mapped to either the ASP's internal local memory or a possible shared memory between the master processor and the ASP(s), while *local and private memories* map to the ASP's fast local memory and general-purpose registers.

Whereas OpenCL standard is designed with Single Instruction Multiple Data (SIMD) or Single Program Multiple Data (SPMD) execution of work-items in mind, our goal was to exploit the instruction scheduling freedom of TTA as much as possible, thus resorting to highly predicated instruction level parallel execution of code from multiple work-items. Thanks to the support of overcommitting resources by means of predicate aware scheduling [16] in TTA it is possible to schedule execution of two operations to the same function unit at the same time instance in case the operations are guarded with opposite predicates. This leads to improved throughput

when compared to SIMD/SPMD style of execution. In SPMD, diverging control flow in the executed work-items usually results in function unit idle time because the branches are executed sequentially.

B. Chaining Work-Items

OpenCL C data parallel execution is described like stream processing: computation on a piece of input data. As the work-items are completely independent from each other, it is straightforward to chain code from multiple work-items by just appending multiple instances of kernel code after each other and allowing the instruction scheduler to parallelize the code between the work-items. The analogy to C-based compilation is to schedule multiple independent iterations of a loop in parallel using loop unrolling or software pipelining. However, an important benefit with OpenCL C kernels is that the basic assumption is that the "loop iterations" (work-items) are independent from each other, in contrast to C loops where complex data dependence analysis is required to prove independence. Figure 2(a) shows a simple OpenCL C kernel structure with a single basic block (a sequence of instructions without branches which is always executed in its entirety). Its original control flow graph (CFG [17]) is shown in Fig. 2(b) and the CFG after work-item chaining and joining to a single basic block in Fig. 2(c). The final form of the code shows that the processor can execute instructions from two work-items in parallel if there are free datapath resources. Another benefit from the chaining is the ability to potentially hide operation latencies due to long latency operations like divisions or memory loads of one work-item with instructions from the another.

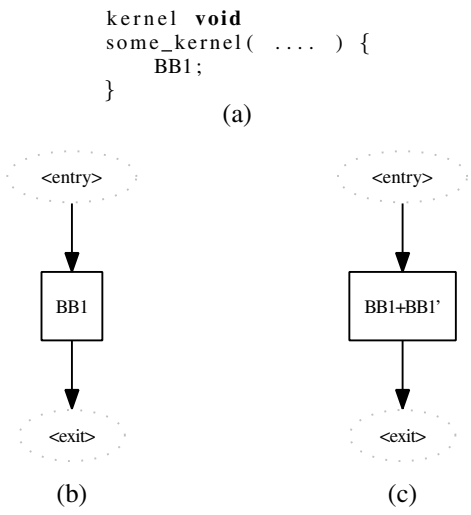

Fig. 2. Simple example on work-item chaining: (a) OpenCL C kernel source, (b) original kernel CFG, and (c) a CFG with two work-items chained and joined.

Some complexity to work-item chaining is introduced by the *work-group barriers*. In the presence of barriers, all work-items in the same work-group are expected to synchronize their execution at the barrier call sites. That is, whenever a single work-item reaches a barrier, it cannot proceed its execution until the rest of the work-items in the work-group have reached it. Thus, in case the work-items are to be chained statically, the kernel has to be split at barrier points and the chaining has to be done with the split parts.

The example in Fig. 3 shows the chaining of two work-items in case of a simple kernel with a barrier call in the middle. In this case chaining is still relatively straightforward: just duplicate and chain the basic blocks before the barrier and connect the last basic block in the copied chain to the "barrier pseudo basic block" (which is just an instruction scheduling barrier in our case) and similarly duplicate and chain the basic blocks after the barrier. In this example, the code before the barrier includes a simple if-else structure. In such case, each control flow structure needs to be duplicated as a whole for each work-item due to the single program counter execution. A succeeding if-conversion [18] pass attempts to convert these control structures to single instruction level parallelizable predicated basic blocks. However, the code after the barrier is a single basic block without branching, thus the chaining algorithm can join the basic blocks of the two work-items to a single one.

When there are barriers inside a conditional basic block or a loop body, the work-item chaining becomes more complex as the problematic nature of static compilation of independent execution using a single program counter becomes more apparent. According to the OpenCL standard, in case of a loop with barriers, each iteration of the loop is synchronized separately. Thus, when a single work-item reaches the barrier in an iteration, it waits for the rest of the work-items to complete the code before the barrier at that iteration. Conversely, when there is a barrier inside a loop, it can be assumed that all work-items execute the loop the same number of times, otherwise the end result is undefined (the barrier causes a subset of work-items to lock up indefinitely). The work-item chaining in this case can be done by treating the loop body independently from the loop construct as is done in [7]. The loop construct is retained as in the original kernel to not break the semantics and the number of iterations in the loop, but the code before and after the potential barriers is duplicated for each work-item.

C. Work-item Chaining Algorithm

The algorithm for statically generating code for every work-item (effectively replicating the kernel code the required number of times) is implemented as a set of closely related LLVM optimization passes. The high-level structure of the whole replication process is shown in Fig. 4.

The first step for the algorithm is to find the barriers, i.e., calls to OpenCL C *barrier()* API function, present in the kernel code. As the barriers do not need to be in the main kernel function code, but might have been placed by the programmer in some of the kernel called sub functions, a prior "flattening" is required. This process performs aggressive function inlining for all non-kernel functions, thus ensuring kernels themselves have no calls once flattened. Apart from easing the barrier detection, flattening also improves the results

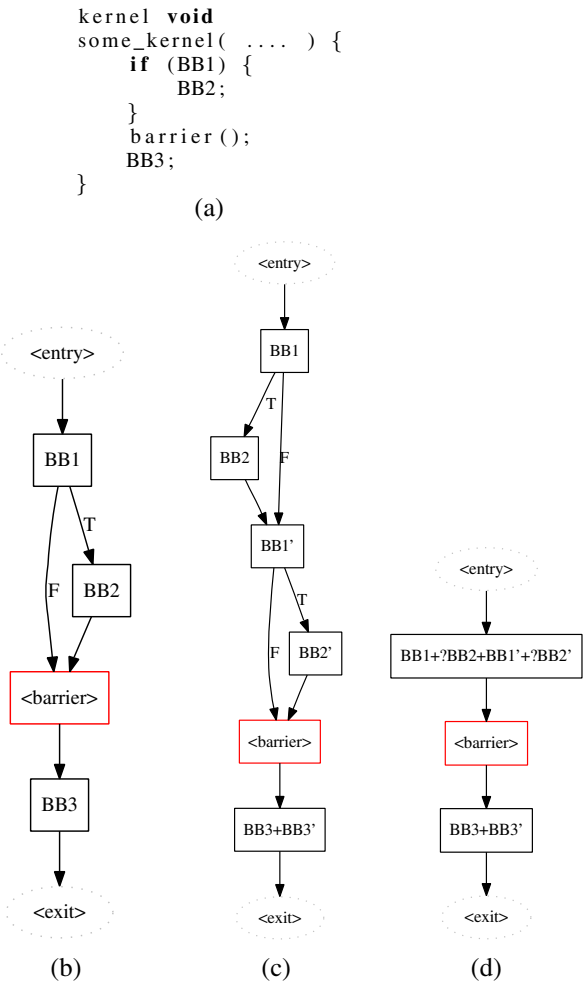

Fig. 3. Work-item chaining with barriers: (a) OpenCL C kernel source, (b) initial kernel CFG, (c) two work-items chained, and (d) the CFG after branching eliminated with if-conversion.

LLVMOPENCL($module$)
1 **for** each Function $f \in module$
2 **do if** IsKernel(f)
3 **then** FLATTEN(f)
4 DETECTBARRIERS(f)
5 **for** each Region $r \in f$
6 **do** LOOPREGION(r)
7 REPLICATECODE(r)
8 **return** $module$

Fig. 4. Work-item code replication algorithm.

of latter language-independent optimization passes such as loop-unrolling or dead code elimination.

Each of the regions between barrier calls are then processed independently. In order to follow the OpenCL programming model, the regions need to be executed a number of times equal to the work-group size. This can be achieved by two different ways: creating loops or replicating the code for each work-item. The former has the advantage of keeping the code size small, but results less ILP to be exploited, while the latter creates more ILP but can lead to huge programs needing lots of resources and processing time to schedule. In order to parametrize this tradeoff, our algorithm uses a runtime parameter to determine the maximum number of replications to be performed per region, thus the number of work items potentially executed in parallel, and generates the remaining work-item executions using loops.

The region replication algorithm works like the basic loop unrolling that is modified to mark instructions belonging to different work-items with an unique annotation to help alias analyzer in recognizing independent instructions. Each basic block is replicated, maintaining the intra-region control flow structure, and an unconditional branch is then added at the end of the previously existing region to ensure the replicated code is run after the original. This process is repeated as many times as required according to the number of parallel work-items to be created.

The region chaining algorithm is designed to generate valid and easy-to-debug code, but it does not perform any optimization. As such, it creates several basic blocks connected by unconditional branches, which can be combined into a single larger basic block. After the region replication has been performed for each kernel, the whole code is linked with the host program and a global optimization stage takes place to reduce this unoptimized code to a smaller and more efficient form.

D. Efficient Instruction Scheduling of Work-Items

The processors generated with our design flow are statically scheduled VLIW-style architectures with up to hundreds of programmer visible general-purpose registers. In order to not hinder the post-pass instruction scheduler from exploiting the potential parallelism between work-items due to "false dependencies" introduced by the reuse of registers, we implemented a customized register allocator. The goal for the register allocator is to assign different registers for the chained work-items to allow them to be fully parallelized.

The register allocator implementation is based on the LLVM version of the Linear Scan Register Allocator [19] by adding a round-robin style bookkeeping for the indices of the registers allocated to variables. This way variables get assigned new registers whenever possible. This simple modification caused the instructions from different work items to usually have registers allocated from different register sets, resulting in a reduced number of register antidependencies. However, this allocation strategy is not even close to optimal due to its greediness that results in more spill code than necessary. Work is ongoing to improve the register allocator to minimize harmful false dependencies while still preserving conservative register usage.

The another source for data dependencies in programs leading to unnecessary sequentialization are the memory accesses. In case the program contains stores, it is not legal to schedule a succeeding load before or parallel with the store unless it can

be proven that the store and the load never access the same memory address. The problem of figuring out whether the same memory location is accessed by two different memory instructions is called "alias analysis".

When scheduling instructions from multiple work-items of OpenCL C kernels in parallel there are several useful properties to assist the alias analysis:

1) All pointer arguments to the kernel function can be assumed to not alias with each other within the work-item. Thus, the pointers can be marked as "restricted pointers" (introduced by ISO C99 [20]) allowing reordering memory accesses to the different input and output buffers within a single work-item.
2) Accesses to the different address spaces cannot alias. That is, even in case the *global* and *local* memories were mapped to the same physical address space, the instruction scheduler can treat them as disjoint areas and reorder the accesses.
3) Accesses through pointers to the constant memory can be assumed to be only reads. Thus, no overlapping with non-const pointers can happen. Furthermore, as the constant memory is known to be truly read-only (contrary to the const pointers in C/C++, for example, which can point to memory that is modified by non-const pointers) no write can alias with constant memory reads.
4) Most importantly: in the regions between work group barriers, the memory accesses of different work-items can be considered not to alias. This allows treating the chained work-items as fully independent regions of code.

The alias analyzer of our instruction scheduler takes advantage of these special properties of OpenCL C to minimize the data dependencies in the work-item chained code, resulting in more scheduling freedom.

E. Custom Operation Support

The use of custom operations, also known as special instructions or special function units (SFUs), is often the most important way to accelerate the execution of an application running in an application-specific processor. The capability to support custom operations without restrictions to their complexity enables gradual optimization of the architecture by adding more and more target-specific custom operations until the performance is close or equal to an accelerator implemented purely as a non-programmable hardware block. Therefore, it is crucial to provide seamless support for programmers to access custom operations from the source code level.

The OpenCL standard defines an API to provide support for vendor specific extensions (see [5], Chapter 9). This API is used in our framework as a means to access the custom operations available in the target processor. The standard requires the OpenCL compiler implementation to generate specifically named preprocessor macros when an extension is supported. In our toolset, the required headers and macros to produce the inline assembly that triggers the custom operations are generated automatically from an architecture description file. Thus, it is possible to compile the same OpenCL C kernel code both to a target that supports and does not support the custom operation in question by using the preprocessor to select the accelerated custom operation or the software-only version. One important benefit from this is that the custom operation accelerated program can be still compiled with a regular GPGPU tool chain like that of NVIDIA's without modifications in case a software version of the custom operation is provided.

```
#ifdef cl_TCE_ADDSUB
 clADDSUBTCE(a, b, c, d);
#else
 c = a + b;
 d = a - b;
#endif
```

Fig. 5. Example of using a custom operation inside an OpenCL kernel in a portable way.

An example code snippet that uses a 2-input-2-output custom operation *ADDSUB*, which adds and subtracts its operands in parallel is shown in Fig.. 5. The *#else* branch executes the same operation in software to maintain portability.

VI. Experiments

In order to validate and measure the performance and feasibility of the use of OpenCL for ASP design in practice, we implemented an Advanced Encryption Standard (AES) encoder using the design flow.

AES uses a data block of 128 bits and a key size of 128, 192 or 256 bits. For our experiment we chose 128-bit key size. The operations involved in the algorithm are substitutions, rotations and permutations, using the 128 bits of data as a 4x4 array of bytes. Many software implementations of the algorithm manage the data to be processed as a buffer of *chars*, and all the operations are done in char size. For minimizing the number of memory accesses we used a variation of the Gladman's implementation [21] that packs each 4 bytes of data in 32 bits unsigned values and uses other similar optimizations in some steps of the algorithm for reducing memory read and write operations.

The algorithm is divided into two steps: key expansion and encryption/decryption. The key expansion takes a 128-bit key and generates a 1408-bit expanded key. This step has to be done only once if the key doesn't change, therefore in our OpenCL implementation we implemented this functionality in the main program.

The encrypt and decrypt steps are done for each block of 128 bits on the source data. These functions were implemented as OpenCL kernels. The encryption kernel receives several parameters from the host side: the global buffer to be encrypted, the expanded key, the buffer to store the results, and the substitution tables needed by the algorithm. Using these parameters and its own global identifier each work-item

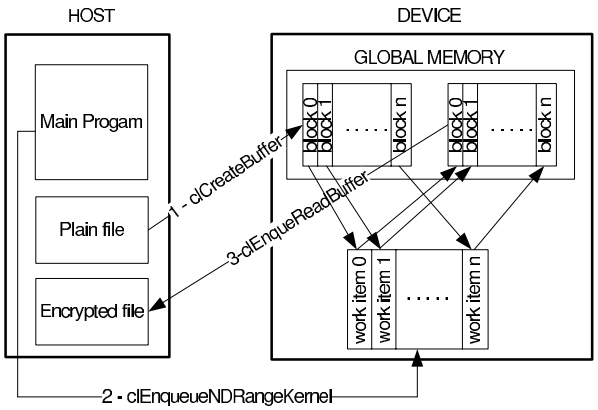

Fig. 6. The OpenCL AES encryption implementation.

Parallel WIs	cycles	speedup
1	35,729	1.00
2	18,209	1.96
4	9,505	3.76

TABLE I
EFFECT OF THE PARALLEL WORK ITEM COUNT TO THE CYCLE COUNT.

executing the kernel calculates the piece of input data it must process.

The host program is responsible for copying the data, key, and substitution tables to the device global memory. Once data is on device memory, the host launches as many work-items as there are 128-bit blocks in the input data buffer that must be encrypted or decrypted, and finally when all the work-items have finished it reads back the results (see Fig. 6).

A. Instruction-level Parallelism

The first experiment was conducted to verify the instruction level parallelism scalability of the OpenCL implementation. In order to measure this, we designed an architecture that provided enough resources for the program to be limited only by its data dependencies. The OpenCL application was compiled for this architecture with one, two and four parallel work items.

The benchmark program encrypted 4KB of random data. The cycle counts with different number of parallel work items are shown in Table. I. The numbers show that the compiler optimizations described in the paper are able to take advantage of the explicit parallelism in the OpenCL kernels, and, given enough resources in the target machine, parallelizing the work items perfectly producing approximately linear speedup with relation to the number of parallel work items.

B. Custom Operations

In the second experiment, we evaluated the use of custom hardware operations to accelerate the application using the OpenCL C extension API as proposed in Section V-E. For this experiment, we designed a realistic base architecture named *AESTTA* with datapath resources as shown in Table II. The connectivity between the datapath units was clustered

resource	multiplicity	notes
Arithmetic-logic unit	3	1 cycle latency
Register file	3	16 registers per file
Load/Store unit	1	2 cycle load latency
32-bit multiplier unit	1	3 cycle latency

TABLE II
RESOURCES IN THE AESTTA PROCESSOR.

architecture	cycles	speedup	KB/s at 100 MHz
AESTTA	1,119,415		366
AESTTA+MUL_GAL	450,490	2.5	909
AESTTA+MUL_GAL+SS	286,778	3.9	1,428

TABLE III
SPEEDUPS FROM CUSTOM OPERATIONS.

VLIW-like with FUs and RFs divided to three one-FU-one-RF clusters. The three clusters were interconnected with a fully connected transport bus.

In order to verify that the architecture is implementable without long critical paths ruining the performance due to low clock frequency, the architecture was synthesized on two FPGA chips: Xilinx Virtex 5 and Altera Stratix II. The maximum clock frequencies were 191MHz for Virtex 5 and 149MHz for Stratix II.

Two custom operations were designed and added to the base architecture:

- *MUL_GAL*, a multiplication of two integers in the Galois field GF(2^8). The software implementation needs two reads from a logarithm table, a read from an antilogarithm table, an addition, and some control for performing this multiplication. In hardware, it can be done in a single clock cycle using two ROMs for the tables, and an 8-bit adder.
- *SUBSHIFT* involves searching in a look-up table, substituting and mixing some elements of an 4×4 array. In software, it takes several clock cycles for reading the look-up table and mixing the elements of the array, but in hardware this operation can be done in a single clock cycle using a ROM and multiplexers.

The same encoding benchmark with the random 4KB input data set as in the previous experiment was compiled with two parallel work items and simulated with the architecture simulator to produce the cycle counts for the kernel execution. The speedups from using the two custom operations in comparison to the software-only AESTTA are shown in Table III. For curiosity, in addition to the cycle count speedups, the table includes the calculated encoding throughput with 100 MHz clock frequency.

The results show that adding custom operations using the extension mechanism works and provides remarkable speedups as expected. Adding both custom operations to the machine produces almost 4x speedup in comparison to the software-only version. By inspecting the generated code, the speedup is partially due to reduced general purpose register

pressure which results in less spills and less antidependencies that constrain the parallelism.

In this case, it would be possible to further accelerate the design with little effort, for example, by adding a fourth cluster to the base machine, increasing the number of general purpose registers, or by adding more custom operations to the design. It can be seen from the previous experiment that given enough resources, the cycle count can be reduced considerably. However, the purpose of this experiment was not to design the fastest possible AES hardware implementation, but to provide a proof-of-concept for the proposed OpenCL-based ASP design methodology.

VII. Conclusions and Future Work

This paper proposes a design methodology that uses the OpenCL standard in application-specific processor (ASP) design. The leading idea in this work is to exploit the ability of OpenCL to describe parallelism at its multiple granularities and exploit the instruction level parallelism of the ASP as much as possible while providing a clean access to custom operations.

In our experiments, we verified that the parallel application description capabilities of OpenCL make it possible to scale the single core performance of the ASP design efficiently by increasing the number of parallel work-items and datapath resources. The custom operation support was verified by using two non-trivial custom operations to accelerate the AES ASP design.

The next steps in our work is to extend and implement the static work-item chaining algorithm to cover more OpenCL kernels with more complex barrier usage scenarios and to improve the efficiency of the instruction scheduler on machines with small number of registers and reduced connectivity. In addition, we plan to add support for generating multicore ASPs to exploit task level parallelism.

Acknowledgment

The authors are thankful for the constructive comments from Dr. Jani Boutellier (Univ. Oulu), Dr. Pertti Kellomäki, Dr. Jari Nikara, Dr. Claudio Brunelli, and Dr. Eero Aho (Nokia).

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *Proc. Annual Int. Symp. Comput. Architecture*, Vancouver, BC, Canada, June 10–14 2000, pp. 248–259.

[2] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *Proc. Annual Conf. Design Automation*, San Francisco, CA, June 15–19 1998, pp. 732–737.

[3] D. M. Ritchie, "The development of the C language," in *ACM SIGPLAN Conf. History of Programming Languages*, Cambridge, MA, Apr. 20–23 1993, pp. 201–208.

[4] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu, "Automatic detection of parallelism: A grand challenge for high performance computing," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 2, no. 3, pp. 37–47, 1994.

[5] *OpenCL Specification v1.0r48*, Khronos Group, Oct. 2009. [Online]. Available: http://www.khronos.org/registry/cl/

[6] T. R. Halfhill, "Parallel Processing with CUDA," *Microprocessor Report*, Jan. 2008.

[7] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Languages and Compilers for Parallel Computing*, ser. LNCS, J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, vol. 5335, pp. 16–30.

[8] A. Papakonstantinou, K. Gururaj, J. Stratton, J. Cong, D. Chen, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *IEEE Symp. Application Specific Processors*, San Francisco, CA, July 27-28 2009, pp. 35–42.

[9] "AutoESL Design Technologies, Inc." http://www.autoesl.com. [Online]. Available: www.autoesl.com

[10] F. Pratas and L. Sousa, "Applying the stream-based computing model to design hardware accelerators: A case study," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. LNCS, K. Bertels, N. J. Dimopoulos, C. Silvano, and S. Wong, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, vol. 5657, pp. 237–246.

[11] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[12] ——, "TTAs: missing the ILP complexity wall," *J. Syst. Architecture*, vol. 45, no. 12-13, pp. 949–973, 1999.

[13] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of transport triggered architectures," in *Proc. Annual Int. Symp. Microarchitecture*, San Jose, CA, Nov. 30–Dec. 2 1994, pp. 191–195.

[14] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia on Mobile Devices*, San Jose, CA, Jan. 29–30 2007, pp. 65 070X–1 – 65 070X–11.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, Palo Alto, CA, Mar. 20–24 2004, p. 75.

[16] M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H.-H. S. Lee, "Predicate-aware scheduling: A technique for reducing resource constraints," in *Proc. Int. Symp. Code Generation Optimization*, San Francisco, CA, Mar. 23–26 2003, pp. 169–178.

[17] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, 1970.

[18] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. ACM SIGACT-SIGPLAN Symp. Principles Programming Languages*, Austin, TX, Jan. 24–26 1983, pp. 177–189.

[19] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM T. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, 1999.

[20] *ISO/IEC 9899:1999 - Programming languages - C*, ISO, Dec. 1999. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG14/www/standards

[21] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin, "Efficient software implementation of AES on 32-bit platforms," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. LNCS, B. S. J. Kaliski, C. K. Koc, and C. Paar, Eds. Berlin, Heidelberg, Germany: Springer-Verlag, 2003, vol. 2523, pp. 159–171.